Aspect: Detecting Bugs with Abstract Dependences

DANIEL JACKSON
Carnegie Mellon University

Aspect is a static analysis technique for detecting bugs in imperative programs, consisting of an annotation language and a checking tool. Like a type declaration, an Aspect annotation of a procedure is a kind of declarative, partial specification that can be checked efficiently in a modular fashion. But instead of constraining the types of arguments and results, Aspect specifications assert dependences that should hold between inputs and outputs. The checker uses a simple dependence analysis to check code against annotations and can find bugs automatically that are not detectable by other static means, especially errors of omission, which are common, but resistant to type checking. This article explains the basic scheme and shows how it is elaborated to handle data abstraction and aliasing.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.2.4 [**Software Engineering**]: Program Verficiation—*assertion checkers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution*; D.3.3 [**Programming Language**]: Language Constructs and Features—*abstract data types*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Reasoning about Programs—*assertions*; *mechanical verification*; *pre- and postconditions*

General Terms: Documentation, Languages, Verification

Additional Key Words and Phrases: Dataflow dependences, partial specification, partial verification

1. INTRODUCTION

Aspect is an attempt to find some middle ground between program verification and type checking. Verification can in theory detect any bug, but for everyday projects its cost is prohibitive. Type checking, on the other hand, is cheap, but catches only the grossest flaws.

Currently, the only economical alternative is testing. But its dynamic nature brings many disadvantages. Execution demands completeness, both in the code—ruling out the analysis of unfinished programs—and in the test cases—requiring full details of a sample input, however simple a property of

This research was supported by DARPA grants N00014-89J-1988 and F33615-93-1-1330, by NSF grants 8910848-CCR and CCR-9308726, and by a grant from the TRW Corporation.
Author's address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: daniel.jackson@cs.cmu.edu.
Permission to make digital/hard copy of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery, Inc. (ACM). To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 1995 ACM 1049-331X/95/0400-0109 $03.50

the output one wants to determine. A static analysis like Aspect manipulates abstract properties of the code, and it can thus provide partial information at lower cost than testing and can be applied to fragments of a program during its construction.

A number of static analyses in this vein have been proposed. All of these, roughly speaking, are based on abstractions of the program state. The state may be abstracted explicitly in a state transition model that is finite [Henderson 1975] or made tractable by approximation [Bourdoncle 1993], or implicitly through a propositional logic [Howden and Wieand 1994; Perry 1989b], a set constraint language [Russell et al. 1994], or a type system [Freeman and Pfenning 1991; Strom 1983]. Section 6 discusses these schemes in more detail.

Aspect is not based on the values of states, but on dataflow dependences between their components. Instead of asserting, like a type declaration, that the poststate of a procedure lies in some set, an Aspect specification asserts a set of dependences that the components of the poststates should bear on the components of the prestates. If a required dependence is missing from the code, then the poststate is computed without adequate information, and there must be a bug. Checking involves little more than the construction of program dependences, so it is automatic, fast, and easy to implement.

The high cost of traditional verification comes not only from the intractability of proofs (which burdens the user with providing lemmas and proof strategies), but also from the difficulty of writing complete formal specifications. Aspect specifications are trivial in comparison and certainly a lot less trouble to write than the code itself. They are not, however, as simple as type declarations.

Although static, in one respect the Aspect analysis is more like testing than verification. In contrast to state-based analyses like type checking, Aspect never reports spurious errors. So when an error is reported, there must be a bug in the code (or, of course, a mistake in the specification), whereas a type error might, for example, indicate a flaw in dead code that will never affect an execution. The flip side of this benefit is that Aspect offers no guarantees of partial correctness: there is no well-defined class of run-time errors that are eliminated by an empty error listing. To put it another way, type checking is sufficient (to rule out run-time type errors) but not necessary; Aspect's checking is necessary but not sufficient.

Luckily, Aspect appears to detect a class of bug that complements the class detected by state-based techniques. It is especially good at catching errors of omission, which plague big projects but usually elude type checking. The empty procedure, *SKIP*, satisfies most type specifications, but no nonempty Aspect specification.

Aspect has been implemented for a programming language that contains most of the features that complicate static analysis: aliasing, polymorphism, exceptions, etc. It relies on strong static typing and works best when a program is structured around abstract types. No large case study has yet been performed. However, a few thousand lines of specification have been written, and bugs have been found in real code.

```
procedure updateAverage (n: int; a, x: real)
%a ← a, n, x
%n ← n
1  s: real
2  s := a × n
3  s := s + x
4  n := n + 1
5  a := s/n
```

Fig. 1. A procedure annotated with Aspect assertions.

This article explains the main ideas of Aspect: the meaning of assertions, how they are checked, and how two vital practical issues—aliasing and data abstraction—are handled. Since the assertion language is tiny, and should anyway be tailored to the programming language, its syntax is conveyed only in passing. Instead, the article uses a simple semantic model to explain the meaning and checking of assertions. This model is sufficient to demonstrate the application of Aspect to an Algol-like language with a module structure for encapsulating abstract types. It should not be too hard to adapt Aspect to any language in this mold (such as Modula-3 or Ada). The full design of an Aspect checker for CLU [Liskov et al. 1981] is described in a technical report [Jackson 1992].

Appendices summarize the Aspect language and define the mathematical symbols used in the article. Sections 2.2, 3.3, and 3.4 formalize and elaborate material discussed in other sections and can be omitted. Readers wanting only a brief overview of Aspect should read Sections 1, 2.1, 2.3, 2.4, and 4.

2. THE BASIC SCHEME FOR STATIC STORAGE

2.1 Simple Dependences

Consider a procedure *updateAverage* (Figure 1) that takes three arguments: n—the number of values in a statistical sample, a—their average, and x—a value to be added to the sample. The procedure should update the average a and the sample size n. Assume for now that all arguments are passed by value/result, so that a change to a formal in the body of the procedure is propagated to the actual on return.

The comment following the header is an Aspect specification. Each line asserts a dependence that should hold between the value of a variable in the poststate (given on the left) and the values of some variables in the prestate (on the right). The first line, for example, says that the value of a after depends on the values of a, n, and x before. Or, equivalently, to calculate the new average, you need the old average, the old sample size, and the sample value.

Suppose we made a mistake in the coding and forgot line 5. The Aspect checker would construct the dependences of this faulty code and display the message:

Missing: a *on* n, x

saying that the required dependences of a on n and x are missing. Another slip might be to write x instead of s on the left-hand side of the assignment on line 3, eliciting the message:

Missing: *a on x*.

In addition to giving the dependences of result variables, an Aspect specification says, implicitly, which variables are modified: any variable for which a dependence is not given is assumed to be invariant. So the specification of *updateAverage* implies that a and n, but not x, are modified. The assertion

$n \leftarrow n$

thus means not only that n after depends on n before, but also that there is an execution in which n is modified. Failure to modify a variable is detected as a bug, so omitting line 4 elicits the error message:

Missing: *modification of n*.

Later, we shall see more interesting bugs that Aspect can detect. But even this trivial example illustrates its key features. First, the specification is partial and could not act as a contract between the implementor and user of the procedure. Nothing about the actual values of the variables is expressed, so Aspect cannot, for example, express the precondition that n be nonzero or that it be incremented exactly by one. Nevertheless, the Aspect specification bears a structural resemblance to a conventional specification, having the assertion $a \leftarrow a, n, x$ in place of a formula like $a' = f(a, n, x)$. The similarity between saying that a after depends on a, n, and x before, and saying that a is some unspecified function of a, n, and x is no accident; it justifies the soundness of the checking method (see Section 2.2).

A second key feature is how straightforward checking is: it involves little more than constructing dependences. This construction is compositional: the checker derives the dependences of compound statements from the dependences of their parts and derives the dependences of primitive statements from the specifications of the procedures they call. In line 3, for example, the checker uses the built-in specification of $+$ to determine that its result depends on its arguments and that its arguments are not modified. More-complex datatypes are no harder to analyze. Like type checking (but unlike symbolic evaluation), Aspect scales easily from integers to editor buffers.

Third, like testing and unlike type checking, the Aspect analysis is inherently nonlocal. The error of writing x for n in line 2 would not be caught, because the missing dependence of a on n is "masked" by the same dependence contributed by line 5. Consequently, Aspect is less likely to find bugs in larger procedures.

Fourth, Aspect's minimal dependences are more like liveness than safety assertions. This is why they tend to be good at catching errors of omission, and why they are not susceptible to the "infeasible-paths" problem of state-based techniques. Infeasible paths may mask bugs by introducing bogus dependences, but will not cause Aspect to produce spurious error messages, since these could only result from underestimating the dependences of the code.

The dependences of an Aspect specification might be interpreted not only as minimal but maximal too. We might then report dependences that are not required by the specification but are found in the code. This analysis, in contrast to the analysis chosen, would generate spurious messages—too many, it seems in practice, to be useful. Nevertheless, reporting apparent modifications to variables that are claimed to be invariant is easy to do and would be a sensible addition to Aspect.

2.2 Formalities of Simple Dependences

A program P has a dependence relation D among its variables

$$D(P): Var \leftrightarrow Var$$

where a pair $(x, y) \in D(P)$ means that the value of the variable x after execution of P depends on the value of y before. Each such pair is a "dependence" of the program P.

What does it mean to say that x depends on y? Representing the behavior of the program P as a function p over some set of program variables a, b, c, etc.

$$p: (a, b, c, \dots) \to (a, b, c, \dots)$$

we say that a variable x depends on a variable y when there are two prestates s and s' that are distinguishable only in their y components and lead, under P, to corresponding poststates having different x components:

$$(x, y) \in D(P) \textit{ iff } \exists s, s'. \forall v \neq y.\ s|v = s'|v \wedge p(s)|x \neq p(s')|x.$$

(Here, $s|v$ means the value of variable v in state s.) In other words, x depends on y if the computation of x uses y.

The Aspect scheme is based on a simple observation: if two functions have different dependences, they cannot be the same. Take one function to be the code, from which dependences are extracted, and the other the specification, whose dependences are given explicitly. Then any discrepancy between these dependence sets indicates that the specification and the implementation are different functions, so the code cannot satisfy the specification.

The dependences of the code are built from the bottom up, starting with the dependences of primitive procedures and assignments. Each compound statement has a rule that gives its dependences in terms of the dependences of its parts. The dependences of a sequence, for instance, are obtained by forming the relational composition of the dependences of its constituent statements:

$$D(S;T) \subseteq D(T) \circ D(S).$$

Note the inequality: dependences cannot be calculated exactly, but an approximation that never omits actual dependences is possible. The calculated dependences of

$$x := x + y;$$
$$x := x - y,$$

for example, will, by the above rule, include (x, y), even though the initial value of y cannot affect the final value of x. The direction of the inequality motivates Aspect's focus on missing dependences. Extra dependences may be artifacts of the calculation and are thus not reliable symptoms of bugs.

The incompleteness of Aspect—its unsurprising failure to catch all bugs—can now be seen to arise from two sources. First is its view of functions purely in terms of dependences. Although functions with different dependences cannot be the same, the converse is not true: different functions may have the same dependences. Second is the dependence calculation itself, which may add bogus dependences.

If-statements and loops introduce control dependences that must be taken into account: from outside a procedure, effects due to control dependences are indistinguishable from dataflow dependences. The statement

```
if b1 then
    b2 := true
    end
```

is semantically equivalent to

$b2 := b1 \textit{ or } b2,$

so it must somehow have the dependence $(b2, b1)$, even though it has no direct dataflow from $b1$ to $b2$. This dependence is accounted for by making every variable modified in either branch of an if-statement dependent on the condition variable.

The modification set of a program P is the set of variables it modifies

$M(P)\colon \mathbb{P}\, Var$

where a variable v is modified if the program has an execution that alters v:

$$v \in M(P) \textit{ iff } \exists s.\ s|v \neq p(s)|v.$$

Modifications are, like dependences, constructed compositionally, and are approximated conservatively:

$$M(S; T) \subseteq M(S) \cup M(T)$$
$$M(\textit{if } b \textit{ then } S \textit{ else } T) \subseteq M(S) \cup M(T)$$
$$M(\textit{while } b \textit{ do } S) \subseteq M(S)$$

The rule for calculating the dependences of an if-statement may now be given:

$$D(\textit{if } b \textit{ then } S \textit{ else } T) \subseteq D(S) \cup D(T) \cup ((M(S) \cup M(T)) \times \{b\})$$

Finally, a loop can be viewed as an infinite nesting of if-statements

$$\textit{while } b \textit{ do } S = \textit{if } b \textit{ then } (S; \textit{while } b \textit{ do } S) \textit{ else SKIP}$$

(writing *SKIP* for the empty statement) which suggests the dependence rule:

$$D(\textit{while } b \textit{ do } S) \subseteq D(S; \textit{while } b \textit{ do } S) \cup D(\textit{SKIP}) \cup ((M(S; \textit{while } b \textit{ do } S) \cup M(\textit{SKIP})) \times \{b\}).$$

The empty statement *SKIP* does nothing; so it has no modifications $(M(\textit{SKIP}) = 0)$, and each variable depends only on itself $(D(\textit{SKIP}) = I)$; so

$D(S;T) = D(T) \circ D(S)$
$D(\textit{if } b \textit{ then } S \textit{ else } T) = D(S) \cup D(T) \cup (M(S) \cup M(T)) \times \{b\}$
$D(\textit{while } b \textit{ do } S) = D(S)^* \cup ((M(S) \times \{b\}) \circ D(S)^*)$
$M(S;T) = M(S) \cup M(T)$
$M(\textit{if } b \textit{ then } S \textit{ else } T) = M(S) \cup M(T)$
$M(\textit{while } b \textit{ do } S) = M(S)$

Fig. 2. Calculating dependences and modifications in alias-free code.

$$D(\textit{while } b \textit{ do } S) \subseteq D(S; \textit{while } b \textit{ do } S) \cup I \cup ((M(S; \textit{while } b \textit{ do } S) \times \{b\}).$$

Expanding with the sequence rule above produces

$$D(\textit{while } b \textit{ do } S) \subseteq (D(\textit{while } b \textit{ do } S) \circ D(S)) \cup I \cup (M(S) \times \{b\})$$

which has the least solution

$$D(S)^* \cup ((M(S) \times \{b\}) \circ D(S)^*)$$

where R^* is the reflexive and transitive closure of the relation R. The first expression $(D(S)^*)$ is the contribution from the direct dataflow; the second holds the control dependences of the modified variables on the loop condition and on the variables that affect it.

Since exact calculation of dependences is not possible, we shall henceforth drop the inequalities and define D and M to be the smallest relations satisfying the inequalities. A summary of the rules for calculating D and M is given in Figure 2. There is no rule for value/result procedure call; the specification of the procedure just takes the place of the code, with appropriate renaming of formals to actuals.

2.3 The Roles of a Procedure Specification

Aside from being a yardstick for judging an implementation, a specification of a procedure also acts as a surrogate, replacing it in the analysis of its callers. There are several reasons to prefer the specification of a called procedure to its code.

First, the code may simply be missing; this should not impede the analysis of callers. Incrementality is a prime goal of Aspect, and it is important to be able to analyze incomplete programs.

Second is data abstraction: the specification can hide the representation of the data, making reasoning about its behavior simpler. This is a central feature of Aspect explained later (Section 4).

The third, and more-subtle reason, has to do with the nature of the dependence approximation. A specification may assert that the procedure has fewer dependences than the checker would have inferred by looking at its code. Using such a specification instead of the code allows more bugs to be detected in the caller. This feature of specification is analogous to nondeterminism in conventional specifications: any particular implementation may have properties that are not required by the specification and should not be relied on by a caller. The analogy is only rough however, since these properties are syntactic artifacts of the code that are not observable in its behavior.

As an illustration, consider another version of *updateAverage* (Figure 3) with the same behavior as before but coded a bit perversely. Instead of just

```
procedure updateAverage (n: int; a, x: real)
%a ← a, n, x
%n ← n
s real
s := a × n
s := s + x
a := s/(n + 1)
n := s/a
```

Fig 3. An implementation with extra, unspecified dependences

adding 1 to n, the new version calculates n by dividing the new sample total by the average. In calculating the dependences of this procedure, the checker, unable to perform the algebraic manipulation

$$s/(s/(n+1)) = n+1,$$

cannot eliminate a and x from the dependences of n, thus effectively deriving the specification

$$a \leftarrow a, n, x$$
$$n \leftarrow a, n, x.$$

Unlike the specification provided by the user, this fails to discriminate between a and x, in which it is symmetrical. Suppose now that we are checking a call *updateAverage* (a, n, x) in which the arguments n and a appear in the wrong order. This bug could never be found given only the code of *updateAverage* since it exhibits identical dependences for the first two arguments.[1] Given the user-provided specification of *updateAverage*, however, the checker can insert only the specified dependences, and the bug may be caught.

Despite the advantages of using a specification provided by the user, sometimes it is more cost-effective to omit specifications of called procedures and allow the checker to generate a "specification" directly from the code. Although this might prevent the detection of some bugs, it does not compromise soundness, and spurious bugs will not be reported. If the code of a called procedure is also missing, the checker can still generate dependences soundly by including all possible dependences of arguments on one another. This approximation can be improved with knowledge of the parameter-passing mechanism. An argument passed by value, for instance, cannot be modified or have dependences on other arguments.

2.4 Structured Dependences: Records and Arrays

The more the dependences of a program reflect its structure, the more likely it is that bugs will be manifested as missing dependences. So in code built with data structures like records and arrays, it makes sense to track dependences on individual components of these structures.

[1] The case is in fact a type error; the first and second arguments coincidentally have different types.

```
type poly = record [terms: array [1..10] of term; size: int]
type term = record [ex, co: int]

procedure addTerm (p: poly, e, c: int)
     %p.terms.@el.ex, p.terms.@el.co, p.size ←
     %      p.size, p.terms.@el.ex, p.terms.@el.co, e, c
     i: int, t: term
     begin
1    i := 1;
2    while i ≤ p.size do
3            t := p.terms[i]
4            if t.ex = e then
5                 t.co := t.co + c
6                 if t.co = 0 then
7                      p[i] := p[p.size]
8                      p.size := p.size − 1
9                 else
10                     p[i] := t
11                     end
12           i := i + 1
13           end
```

Fig. 4. An example of structured dependences.

To see how this is done, consider a procedure that adds a term cx^e to a polynomial p represented as an array of exponent/coefficient pairs (Figure 4). To simplify the code, we have assumed (as a precondition) that a term with the given exponent is already present, so the procedure just has to search for this term and update its coefficient. Also, the array is statically allocated, so only polynomials with 10 or fewer terms are allowed. The number of terms in the array is stored explicitly; any array elements with index above *p.size* are ignored. If the new coefficient is zero, the term is removed by replacing it with the last term and decrementing *p.size*.

The names that appear in the dependency assertions now identify components of the data structures rather than plain variables. Since specifying dependences of array elements according to their indices would require the values (and not just the dependences) of other arguments, a single name, *p.terms.@el*, is used to denote the set of array elements. A name like *p.terms.@el.co* then refers to the set of coefficient components of the elements of the array *p.terms*. The namespace simply mirrors the syntax of the type declaration, with one name for each node in the abstract syntax tree. This naive treatment of arrays works fine, so long as the specifier remembers that a single name represents many elements: the assignment update $a[i] := e$, for instance, must be specified as

$$a.@el \leftarrow a.@el, e, i$$

with the *a.@el* on the right to account for the elements that are unmodified.

The single dependence assertion of *addTerm* is a shorthand, collecting the common dependences of all the names on the left into a single assertion. It says that each component of the p structure after depends on all the components before and on the arguments e and c.

There are so many reads and write of the polynomial structure that a dependence analysis based on variables alone would be hopeless. Almost any error would be masked by a surplus dependence. But with the finer-grained analysis, a variety of bugs can be detected. Here are a few: omitting line 5 or writing c on the left instead of $t.co$ (causing missing dependences on c); omitting line 8 (causing missing dependences of $p.size$); and writing $t.ex$ for $t.co$ in the condition of line 6 (causing missing dependences of $p.size$ and $p.terms.@el.ex$ on $p.terms.@el.co$).

Calculating dependences for components is almost identical to regular dependence analysis, as if each component name were a variable in its own right. The only slight complication is that the types of variables must be available to accommodate copying operations that mention only the names of whole structures. The assignment $t := p.terms[i]$, for example, gives dependences of $t.co$ on $p.terms.@el.co$ and $t.ex$ on $p.terms.@el.ex$.

2.5 Making It Practical

The scheme we have outlined so far is feasible, but it has two serious drawbacks, both evident in the *addTerm* example (Figure 4).

The first is the completely static model of storage. With only fixed arrays, we cannot represent all polynomials. Some kind of dynamic allocation is needed. In imperative programs, this inevitably leads to another problem: aliasing. When objects can be created at run-time, there are no longer enough names in the program text to refer to them. This shortage of names is remedied in two ways. One is to reuse the names of declared variables. The other is to create names at run-time; names become values in their own right and can be passed and stored. This is common in object-oriented programs, where the value of an object frequently includes names of other objects. Either way, the dissolution of the rigid link between names and storage locations wreaks havoc for dependency analysis. A syntactic name may no longer refer to the same location throughout the execution of the program, and, worse, at some point, two names may refer to the same location so that a modification under one name may surreptitiously appear to cause a modification under the other too. This is aliasing.

Section 3 shows how aliasing is handled. A new model defines dependences between locations rather than names. The checker calculates the possible bindings of locations to names in the context of a procedure call and can thus determine which locations are affected. To keep track of the naming of objects, a kind of abstract interpretation is performed. The specifications of procedures must be elaborated too, so that the analysis of a client can include the changes a called procedure may bring about in the binding of names to locations.

Aspect can handle dynamic allocation, but it is not discussed here; see Jackson [1992] for details.

The second drawback of our scheme is the lack of abstraction. The specification of *addTerm* exposes the structure of the *poly* type. Any client of *addTerm* with a polynomial as an argument would show this structure in its

specification. Changing the representation of the polynomial under these circumstances is catastrophic; the specification of every procedure that has a polynomial as an argument might need to be rewritten.

Section 4 addresses this second problem by showing how to separate the concrete components of objects from the names used in specifications. Abstract components of objects called *aspects* are introduced. The user chooses the aspects of an abstract type and writes specifications of its procedures in terms of the aspects. Thus clients never see the actual structure exposed in specifications. To check the abstract procedures themselves, the checker translates the concrete dependences into aspect dependences using an abstraction function given by the user.

Aspects help in other ways too. Specifications are much reduced in size and complexity, since a single aspect corresponds often to several concrete components. Also, the abstraction function can describe redundancy in the representation, which might otherwise have required disjunction to be added to the dependence assertion language.

3. A DYNAMIC STORAGE MODEL

3.1 Dependence Analysis For Pointers

To see why aliasing complicates dependence analysis, consider a type that represents a bank customer, consisting of a name and a reference to an account:

$$type\ customer = record[nam\colon name,\ acc\colon \uparrow account]$$
$$type\ account = record\ [id,\ bal\colon int]$$

Now suppose we have two customers, a and b, and we execute an operation that causes them to share accounts:

a, b: *customer*

...

$a.acc := b.acc$

A subsequent operation on a's account will now affect b's and vice versa; they are the same account, stored in the same location. A deposit of x to the shared account

$$a.acc\uparrow.bal := a.acc\uparrow.bal + x,$$

for example, will be seen by both parties even if performed only under the name a. Calculating the dependences of this statement by the simple method, we will obtain, correctly, a dependence of $a.acc\uparrow.bal$ on its old value and on x, the amount added, but we will miss the dependence of $b.acc\uparrow.bal$ on x. If this dependence were required in a specification, the checker would issue a spurious message saying it is missing, when in fact it is present. Aspect's soundness relies on always overestimating dependences, so losing dependences will undermine it and must be avoided.

One solution to this problem is to assume that all locations referenced by pointers of the same type may be aliased. But this is too drastic; it would add far too many dependences and diminish the chance of catching bugs in programs that make liberal use of pointers. A better solution, and the one adopted by Aspect, is to determine the possible aliases according to context, so that whether the statement incrementing $a.acc\uparrow.bal$ affects $b.acc\uparrow.bal$ depends on what was executed previously.

The *context* of a statement gives the binding of names to locations whenever that statement is executed. Of course, at run-time a single statement may be executed in different contexts—for example, if it follows an if-statement that switches pointers—so the calculated context of a statement must account for several possibilities. The checker approximates contexts, like the dependence calculation, in a conservative fashion, so that it guarantees never to underestimate the potential aliasings of names.

One could calculate dependences between names, as before, by maintaining an aliasing relation on names and adding extra dependences to aliased names. But this is unnatural and gets very tricky. When a and b are aliased, the behavior of an assignment to a is symmetrical in its effect, so there is no reason to regard a or b as primary.

It makes more sense to view dependences as relations between *locations*. The checker uses the context of a program statement to translate its effect in terms of names into dependences between locations. To check a procedure's dependences against its specification, the checker again translates the names in the specification into locations and finds their dependences accordingly. Aliasing is thus implicit and handled as a by-product of the translation of names to locations.

Pointers cause surreptitious dependences not only by aliasing but also in the very naming of locations. Suppose that our bank, being old-fashioned, will only allow customers with the same last name to share accounts. Each customer has a first and last name

type name = record [first, last: string],

and the assignment to the pointer *a.acc* is now conditional:

if a.nam.last = b.nam.last then
  a.acc := b.acc
  end.

The context of a subsequent operation will give two possible bindings for *a.acc*: to its initial location, for the case in which the condition is false, and to the location of *b.acc*, for the case in which the condition is true. But something more subtle must be captured too. When

$$a.acc\uparrow.bal := a.acc\uparrow.bal + x$$

is subsequently executed, whether the value of $b.acc\uparrow.bal$ is altered depends on whether a names the same location as b and thus on the initial values of *a.nam.last* and *b.nam.last*.

To account for these *naming dependences*, the checker keeps not only the values of pointers (that is, which locations they might refer to), but also their dependences. Then, when calculating the dependences of a location referenced through other locations, the checker incorporates their dependences. So, in this case, the if-statement would give dependences of *a.acc* on *a.nam.last* and *b.nam.last*, which would be passed on to *a.acc*↑*.bal* when it is subsequently modified.

3.2 Specifications of Procedures that Alter Pointers

So far, we have only considered how pointers complicate the calculation mechanism. The specification language must be elaborated too. To see why, suppose the statement that established the sharing of accounts were embedded in a procedure, *joinAccounts*, of its own. Then to analyze the code

joinAccounts (*a*, *b*)
a.acc↑*.bal* := *a.acc* ↑ *.bal* + *x*

the checker would infer the context effects of *joinAccounts* from its specification. Of course, if the code of the procedure were available, the checker could use that, but for the reasons given in Section 2.3, a modular analysis using the specification of *joinAccounts* would be preferable. Moreover, we shall see that changes to contexts can be checked like dependences, so asserting them in a specification brings more opportunities for detecting bugs.

A new kind of assertion is thus called for. The *reconfiguration* assertion

a.acc :- *b.acc*

says that the location named by *a.acc* after execution of the procedure is any of the locations named by *b.acc* before. To express several possibilities, more than one name may be written on the right, so

a.acc :- *a.acc*, *b.acc*

says that, after execution, *a.acc* names a location called *a.acc* or *b.acc* before. Reconfigurations can be thought of as specifying values as opposed to dependences, and so this one could be read: "the value of *a.acc* after is the value of *a.acc* or *b.acc* before." Aspect shuns the specification and checking of values generally but treats pointers as a special case.

What determines the choice can be expressed with a standard dependence assertion, so *a.acc* ← *a.nam.last*, *b.nam.last* says that the value the pointer *a.acc* takes depends on *a.nam.last* and *b.nam.last*.

To check a reconfiguration assertion, the procedure is executed in an abstract fashion, and a final context is determined. The set of locations bound to the name on the left of the assertion in this final context is then compared to the set of locations bound to the names on the right of the assertion in the initial context. The former set must then include the latter. Equality is not required, because the context calculation may overestimate bindings. Any bindings in the latter but not the former represent possible assignments of pointers that are required, but not evidenced in the code, and are reported as errors.

```
type customer = record [nam: name, acc: ↑account]
type name = record [first, last: string]
type account = record [id, bal: int]

procedure mergeAccounts (a, b: customer)
%c a.acc -a.acc, b.acc
%c a.acc ← a.nam last, b.nam.last
%c b.acc↑.bal ← b.acc↑.bal, a.acc↑.bal
1  x: int;
2  if a nam last = b nam last then
3        x = a.acc↑ bal
4        a.acc = b.acc
5        a acc↑.bal = a.acc↑.bal + x
6        end
```

Fig. 5. A procedure with dependence and reconfiguration assertions.

Since a procedure may alter the bindings of names to locations, a nasty dilemma arises. When we write an assertion like

$$a.acc\uparrow.bal \leftarrow a.acc\uparrow.bal,\ x$$

we might wonder which location is meant by *a.acc* in the expression *a.acc↑.bal*, the location in the initial state or the location in the final state? Does *a.acc* even refer to the same location on both sides of the assertion? The solution adopted by Aspect is to regard every location name in the specification, wherever it appears, as interpreted in the initial context. Without such a rule, there would be a risk that a series of assertions like

$$x\uparrow :\text{-}\ y\uparrow$$
$$y\uparrow.c\uparrow :\text{-}\ x\uparrow.c\uparrow$$

might be construed operationally, with the meaning of *x↑.c↑* in the second assertion affected by the constraint on *x↑* in the first. Aspect is declarative; so the order of assertions is immaterial, and the meaning of an expression does not depend on where it is placed in a specification.

Figure 5 shows an annotated specification of a procedure that joins the accounts of two customers *a* and *b* (on condition that they share the same last name), so that they share the account that was originally just *b*'s, and then credits the joint account with the balance of *a*'s old account, now no longer accessible.

The specification has three assertions. The first, a reconfiguration, says that the account associated with customer *a* after execution may be the account before, or the account associated with *b*. The second, a dependence assertion, says that whether this change occurs depends on the last names of the customers. The third, a dependence assertion too, says that the value of the account originally associated with *b* may be updated by the balance of the account originally associated with *a*.

Let us now consider a few bugs. Suppose we forgot to credit the balance of *a*'s original account to the joint account, omitting statements 3 and 5. The checker would then report

Missing: *b.acc↑.bal on a.acc↑.bal.*

Exactly the same message would be given if we swapped statements 3 and 4 so that the code read

$a.acc := b.acc;$
$x := a.acc\uparrow.bal;$

this time because the balance assigned to x is the balance of the wrong account, the preceding assignment causing $a.acc\uparrow.bal$ to refer to the balance originally called $b.acc\uparrow.bal$. Omitting line 4 not only gives a missing dependence message for the failure to update $b.acc\uparrow.bal$, but also a missing reconfiguration message

Missing: *final a.acc omits initial b.acc*

saying that the value in the final context of the location called *a.acc* (in the initial context) cannot hold the value in the initial context of the location called *b.acc*. Always performing the merge (instead of doing the test), on the other hand, omits the case in which *a.acc* is unchanged, giving

Missing: *final a.acc omits a.acc*

as well as messages saying that the dependences of *a.acc* on the last names are missing. Testing the wrong condition (for example comparing $a.acc\uparrow.id$ and $b.acc\uparrow.id$) would give instead

Missing: *a.acc on a.nam.last, b.nam.last.*

Contexts thus bring not only complication but also the opportunity to detect a new class of error—namely, failures to set pointers correctly.

3.3 Formalities of Contexts and Location-Based Dependences

The notion of dependence must be revised to handle aliasing. Previously, dependences related variables, or in the more-general case, names of locations. The association of names to locations was fixed, so there was never a need to mention locations explicitly. But if the name of a location may change, we must define dependences and modifications over locations instead:

$deps: Loc \leftrightarrow Loc$
$mods: \mathsf{P}\, Loc.$

A location may correspond to an address on the stack or the heap. It actually makes no difference, since Aspect checking is always within the scope of a single procedure call.

A given statement's dependences can no longer be inferred from its syntax alone, since the names the statement uses can denote different locations, depending on the previous statements. Instead, the dependences are derived in a context that binds names to locations.

A context is an abstraction of a set of program states. It consists of an environment and a store:

$Context = Env \times Sto$
$Env = Var \rightarrow Place$
$Sto = Loc \leftrightarrow Place$
$Place = Loc + (Field \leftrightarrow Place).$

The environment maps each variable to a "place," which is either simply a location or, if the variable has compound type (such as record or array), an association of field names and places. The store maps each location corresponding to an expression of pointer type to a place.

Contexts are a static, textual notion. There is one context for each statement in the program—that is, for each value of the program counter—however many times the statement is executed. So a single context must account for all the possible bindings of names to locations under which a statement may be executed. This is why, in general, there will be more than one place associated with a location in the store: the store is a relation and not a function. Places are likewise relations, because an array is represented as a set of locations all labeled with the same field *@el*. The environment, in contrast, is functional, since, for Algol-like languages, the naming of locations by variables is fixed.

Only location values are mapped in the store. Tracking nonlocation values such as integers would be pointless, since these play no role in Aspect checking.

A context for a program with only one variable

c: *customer*

whose type is given, as before, by

type customer = *record* [*nam*: *name*, *acc*: ↑*account*]
type account = *record* [*id*, *bal*: *int*]
type name = *record* [*first*, *last*: *string*]

is shown in Figure 6. Note that there are five locations (depicted as solid boxes): two for the integers, two for the strings, and one for the pointer. With arbitrary names for locations, the environment corresponding to this context is

$$\{c \mapsto \{nam \mapsto \{first \mapsto loc1,\ last \mapsto loc2\},\ acc \mapsto loc3\}\},$$

and the store is $\{loc3 \mapsto \{bal \mapsto loc4,\ id \mapsto loc5\}\}$.

To find the locations associated with an expression, we simply follow through the environment and then, repeatedly for each level of indirection, through the store. Each expression naming a location is viewed as a sequence

$$Expr = Var \times (Field + Ptr)^*$$

where *Ptr* is the pointer dereferencing symbol. The elements of the sequence are looked up in place mappings or the store, depending on whether they are fields or pointer indirections. The expression *c.acc↑.bal*, for example, is regarded as the sequence

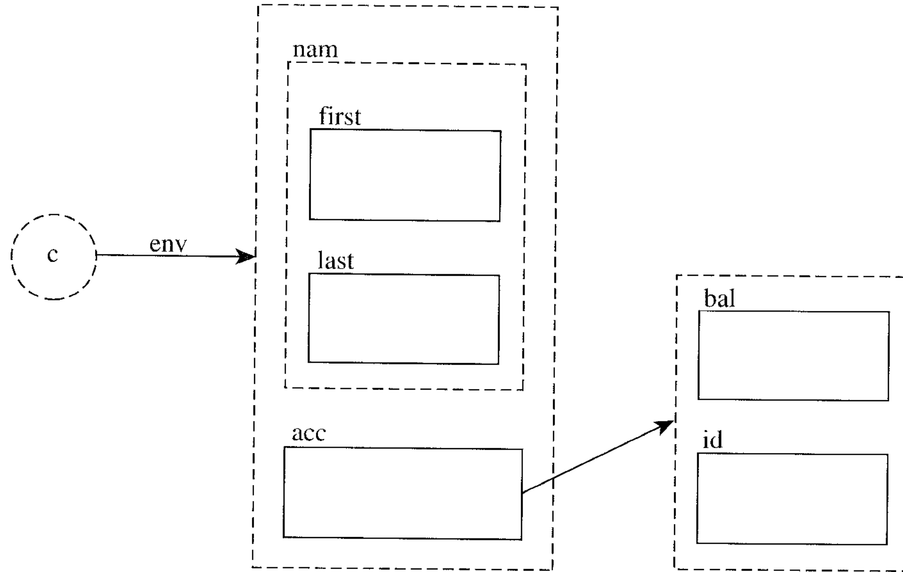

Fig. 6. An example of a context.

$\langle c,\ acc,\ \uparrow,\ bal\rangle$.

We look up c in the environment, giving a place in which we look up the field *acc*. This in general will give a set of locations (named by the expression *c.acc*) which are then looked up in the store, to obtain a set of places, in which we look up the field *bal*, finally obtaining the set of locations denoted by the entire expression.

We shall factor out the naming of locations by defining a function $Binds_c$, that, for each context C, maps each expression to the set of locations it might refer to in C:

$$Binds_c\colon Expr \to \mathsf{P}\, Loc$$
$$Binds_c(\langle v\rangle \frown fs) = F(fs,\ C.Env(v))$$
$$F\colon (Field + Ptr)^*,\ Place \to \mathsf{P}\, Loc$$
$$F(\langle\rangle, p) = \{p\}$$
$$F(\langle f\rangle \frown fs, p)$$
$$= \bigcup\{F(fs, p') \mid \mathit{if}\ p \in Loc\ \mathit{then}\ p' \in Sto[\{p\}]\ \mathit{else}\ p' \in p[\{f\}]\}$$

(Square brackets denote relational image: $R[S]$ is the set of values associated with the set S in relation R.) This function will only be applied to expressions that denote locations. A statement such as $a.nam := b.nam$ will thus be viewed as an abbreviation for

$$a.nam.first := b.nam.first$$
$$a.nam.last := b.nam.last,$$

and the question of which locations are associated with $a.nam$ will never arise.

We will also have use for the set of expressions that, for a given context, denote more than one location:

$$Unsures_c : \mathbb{P}\, Expr$$

$$Unsures_c = \{e \in dom(Binds_c) \mid \#Binds_c(e) > 1\}.$$

When an assignment is made to a pointer variable, the value of its location is replaced in the store by some new value. Since this location's value may, through further dereferencing of fields and pointer indirection, lead to further locations, the *value* of one expression can determine which locations another refers to. Altering the value of the expression *c.acc*, for instance, will give *c.acc*↑*.bal* a new set of locations.

When we look at the dependences associated with an expression, we must therefore include not only the explicit dependences of the locations it refers to, but also the dependences of the locations that determined *which* locations it should name. Since these "stepping-stone" locations are given by expressions that are prefixes of the expression in question, we will have use, when we formalize this (in Section 3.4), for a relation

$$Prefix : Expr \leftrightarrow Expr$$

with $(r, s) \in Prefix$ when s is a location-valued expression that is a prefix of r.

To see how this all works, consider some contexts that might arise in the execution of the if-statement from our example (Figure 5):

```
if a.nam.last = b.nam.last then
    x := a.acc↑.bal
    a.acc := b.acc
    a.acc↑.bal := a.acc↑.bal + x
    end.
```

Suppose the initial context has the environment

$$a \mapsto \{nam \mapsto \{first \mapsto loc1,\ last \mapsto loc2\},\ acc \mapsto loc3\}$$

$$b \mapsto \{nam \mapsto \{first \mapsto loc11,\ last \mapsto loc12\},\ acc \mapsto loc13\}$$

$$x \mapsto loc20$$

and the store

$$loc3 \mapsto \{bal \mapsto loc4,\ id \mapsto loc5\}$$

$$loc13 \mapsto \{bal \mapsto loc14,\ id \mapsto loc15\}.$$

After the assignment *a.acc* := *b.acc*, the environment will be the same, but the store will have a new value for *loc3*, equal to that of *loc13*:

$$loc3 \mapsto \{bal \mapsto loc14,\ id \mapsto loc15\}$$
$$loc13 \mapsto \{bal \mapsto loc14,\ id \mapsto loc15\}.$$

Locations *loc4* and *loc5* are now inaccessible. The next statement's mutation of *a.acc↑.bal* affects *loc14*, so its dependences will include, for example, *loc14* on *loc20*.

Since the if-statement might not be executed, the context following it must incorporate the two possible values of *a.acc*, so its store will map *loc3* to two places:

$$loc3 \mapsto \{bal \mapsto loc4,\ id \mapsto loc5\}$$
$$loc3 \mapsto \{bal \mapsto loc14,\ id \mapsto loc15\}$$
$$loc13 \mapsto \{bal \mapsto loc14,\ id \mapsto loc15\}$$

giving the unsure expressions {*a.acc↑.id*, *a.acc↑.bal*}. What locations these expressions denote depends on the value of *a.acc* since

$$(a.acc\uparrow.id,\ a.acc),\ (a.acc\uparrow.bal,\ a.acc) \in Prefix.$$

Ultimately, each of these will contribute a naming dependence; for example, *loc5* (the location originally called *a.acc↑.id*) will depend on *loc3* (the location originally called *a.acc*).

3.4 Semantics of Specifications in Contexts

Before we introduced pointers, specifications expressed dependence relations directly and just had to be instantiated appropriately at any given call site. Now, however, dependences are between locations that do not have a fixed correspondence to the expressions that appear in a specification. The dependence relation of a procedure P thus depends on the context c in which it is invoked:

$$D(P, c)\colon Loc \leftrightarrow Loc.$$

We must therefore explain how a specification S, which has syntactic dependences D_s

$$D_s\colon Expr \leftrightarrow Expr,$$

is translated into a semantic dependence relation

$$D(S, c)\colon Loc \leftrightarrow Loc.$$

There is similarly a distinction to be made between the locations that a procedure with specification S modifies in a context c

$$M(S, c)\colon \mathrm{P}\, Loc$$

and the specified modification set, given implicitly as the domain of the dependence relation:

$$M_s\colon \mathrm{P}\, Expr$$
$$M_s = dom(D_s).$$

If a procedure with specification S is called in context c, the modifications are obtained by using c to translate the syntactic location names into locations:

$$M(S, c) = \bigcup \{Binds_c(m) \mid m \in M_s\}.$$

Dependences are harder to calculate. First we define a function to translate a relation over expressions into a relation over locations for a given context C:

$$Trans_c : (Expr \leftrightarrow Expr) \rightarrow (Loc \leftrightarrow Loc)$$

$$Trans_c(D) = \{(x, y) \mid \exists (e, f) \in D.\ x \in Binds_c(e) \wedge y \in Binds_c(f)\}.$$

Then the dependences are obtained as follows:

$$D(S, c) = I_{Loc} \oplus Trans_c\big(D_s \cup I_{Ms \cap Unsures_c} \cup (D_s \cup I_{Ms}) \circ Prefix\big).$$

The identity relation on locations appears as a frame condition, so that a location depends on itself unless explicitly overridden. The main term gives the explicit dependences as the translation of three relations over expressions into a relation over locations. The first relation contains the directly specified dependences. The second makes the modified locations whose naming is unsure explicitly dependent on themselves, since any such location might not in fact be modified.

The third adds naming dependences: if a location is altered under some name, its subsequent value must depend on the locations that determined the names under which it was altered. A dependence assertion $e \leftarrow f$ will have naming dependences of the locations denoted by e on the locations denoted by prefixes of both e and f; this is why the identity relation on modified expressions and the dependence relation are taken together.

The inclusion of these naming dependences (and no more) reflects a fundamental assumption that locations can be accessed only by name. Dependences introduced by other routes will be missed (and might lead to spurious bug reports). A daring C programmer, for example, might rely on the colocation of two fields of a struct and access the second by incrementing a pointer to the first. This is not expressible in our model.

So far, the meaning of a specification is, as before, a set of dependences and implicit modifications, still defined over expressions (previously only variables) but whose effects in terms of locations depend on the context. An entirely new effect of a procedure is to change the context itself. To express this, we add a "reconfiguration relation" to the specification, which, like the dependence specification, relates expressions

$$R_s : Expr \leftrightarrow Expr$$

but has a different meaning. It describes possible values, rather than dependences, of the locations. We write r :- s to include the pair (r, s), which says that the postvalue of the location called r might be one of the prevalues of the

location called s. As with dependences, the omission of an assertion for a name implies that its value is unchanged.

Since the naming of locations is itself changing here, we have to pick a context in which r and s are themselves interpreted. This is the precontext—the context when the procedure is called—as for the interpretation of D_s and M_s. The new context resulting from calling a procedure with specification S in context c is obtained by translating R_s into locations and composing with the store:

$$C(S, c) = \left(Env_c, \left(I_{Loc} \oplus Trans_c(R_s \cup I_{Ms \cap Unsures_c})\right) \circ Sto_c\right).$$

As before, locations that are not mentioned are unchanged (because of the frame condition given by the identity on locations), and locations named by unsure expressions retain, in addition to any new bindings, their old values (because of the identity relation on expressions).

Since the reconfiguration (r, s) involves a dataflow from the location called s to the location called r, there must also be a dependence, and so any specification must satisfy

$$R_s \subseteq D_s.$$

Rather than requiring the specifier to add the dependences associated with R_s explicitly, we take the reconfiguration assertion r:-s to include the dependence of r on s. (Figure 7 collects together all the definitions of the dynamic model introduced so far.)

3.5 Checking with Contexts

Introducing contexts to handle pointers complicates the checking mechanism considerably. A statement's dependences cannot be calculated until its context is known, so the checker must construct contexts and dependences hand-in-hand. The final result of evaluating the code of a procedure is now not only a dependence relation but also a final context. The context is used, with the initial context, to check the reconfiguration assertions.

Contexts add not only complications but also new opportunities to detect errors. By comparing the initial and final contexts, the checker can catch aliasing errors. Each reconfiguration assertion is checked against the initial and final contexts, and if a location in the final context fails to contain all possible values dictated by the specification, an error message is displayed.

The rules for constructing contexts, dependences, and modifications from code (Figure 8) are structured as before but with contexts added throughout. Note how the modifications or dependences of two statements in sequence can no longer be calculated independently. The first rule for modifications, for instance, shows how the modifications of the second statement T depend on the context resulting from the execution of its predecessor S. For loops, the modifications of the body may change on each iteration.

The recursive rules for loops are to be interpreted as defining the smallest relation that satisfies the equation. The form of the rule represents accurately how the checker performs the calculation. The dependences, for example, are obtained by unwinding the loop, adding pairs to the relation. The

$D\colon Loc \leftrightarrow Loc$
$M\colon \mathsf{P}\, Loc$

$Context = Env \times Sto$
$Env = Name \leftrightarrow Loc$
$Sto = Loc \leftrightarrow Env$

$Expr = Name^{+}$
$Binds_c\colon Expr \rightarrow \mathsf{P}\, Loc$

$Binds_c(\langle v \rangle \frown fs) = F(fs, C.Env(v))$

$F\ (Field + Ptr)^{*},\ Place \rightarrow \mathsf{P}\, Loc$
$F(\langle\rangle, p) = \{p\}$
$F(\langle f \rangle \frown fs, p) = \bigcup\{F(fs, p') \mid if\ p \in Loc\ then\ p' \in Sto[\{p\}]\ else\ p' \in p[\{f\}]\}$

$Unsures_c.\ \mathsf{P}\, Expr$
$Unsures_c = \{x \in dom(Binds_c) \mid |Binds_c[x]| > 1\}$

$D_s\colon Expr \leftrightarrow Expr$
$M_s\colon \mathsf{P}\, Expr$
$M_s = dom(D_s)$

$M(S, c)\colon \mathsf{P}\, Loc$
$D(S, c)\colon Loc \leftrightarrow Loc$

$M(S, c) = \bigcup\{Binds(m) \mid m \in M_s\}.$

$Trans_c\colon (Expr \leftrightarrow Expr) \rightarrow (Loc \leftrightarrow Loc)$
$Trans_c(D) = \{(x, y) \mid \exists (e, f) \in D \vee x \in Binds_c(e) \vee y \in Binds_c(f)\}$

$D(S, c) = I_{Loc} \oplus Trans_c(D_s \cup I_{M_s \cap Unsures(c)} \cup (D_s \cup I_{M_s}) \circ Prefix)$

$R_s\colon Expr \leftrightarrow Expr$
$R_s \subseteq D_s$

$C(S, c) = (Env_c, (I_{loc} \oplus Binds_c[R_s \cup I_{M_s \cap Unsures(c)}]) \circ Sto_c)$

Fig. 7. Collected definitions for a dynamic model.

$D(S;T, c) = D(T, C(S, c)) \circ D(S, c)$
$D(if\ b\ then\ S\ else\ T, c) = D(S, c) \cup D(T, c) \cup (M(S, c) \cup M(T, c)) \times \{b\}$
$D(while\ b\ do\ S, c) = D(if\ b\ then\ (S;\ while\ b\ do\ S)\ else\ SKIP, c)$
$D(SKIP) = I_{loc}$

$M(S;T, c) = M(S, c) \cup M(T, C(S, c))$
$M(if\ b\ then\ S\ else\ T, c) = M(S, c) \cup M(T, c)$
$M(while\ b\ do\ S, c) = M(S;\ while\ b\ do\ S, c)$
$M(SKIP) = \emptyset$

$C(S;T, c) = C(T, C(S, c))$
$C(if\ b\ then\ S\ else\ T, c) = C(S, c) + C(T, c)$
$C(while\ b\ do\ S, c) = c + C(S;\ while\ b\ do\ S, c)$
$where\ c1 + c2 = (Env(c1), Sto(c1) \cup Sto(c2))$
$C(SKIP, C) = C$

Fig. 8 Rules for calculating dependences, modifications, and contexts.

context following the whole loop is calculated simultaneously. When an iteration is reached in which no further pairs are added to the dependence relation, and no further bindings to the context, the calculation is terminated. This fixed point must be reached eventually since there are only a finite number of names and thus a maximal context and dependence relation. In practice, two or three iterations suffice.

The context itself is calculated in the obvious way. When some point in the code can be reached by more than one path, its context must account for the bindings from both paths. So, for an if-statement, the resulting context is the merge of the contexts resulting from the two branches. The merging operator preserves the soundness of the approximation by ensuring that if contexts c_1 and c_2 represent sets of actual program states Σ_1 and Σ_2, then the merged context $c_1 + c_2$ represents a superset of $\Sigma_1 \cup \Sigma_2$.

The environment of a context changes only at new variable declarations; the details are straightforward and have been omitted.

The rules for primitive statements (not shown in Figure 8) depend on the programming language. A copying assignment $e := f$ (as in Pascal) has the specification $e \leftarrow f$ if e and f are expressions of primitive type (e.g., integer) and $e :\!\text{-}\, f$ if they have pointer type. If e and f have record type, the assignment is regarded as a shorthand for a set of assignments $e.c := f.c$ with specifications $e.c \leftarrow f.c$ or $e.c :\!\text{-}\, f.c$ where each c is a sequence of field names that dereferences x and y to nonrecord variables.

Procedure specifications cannot be used directly in evaluating procedure calls, because the context in which the specification is evaluated is not the same as the context in which the procedure is called. An intermediate context has to be generated to describe the binding of formals to actuals. Similarly, there is an intermediate context between the final context given by the specification and the context after the return of the procedure call. The procedure call is thus divided into three phases: the *call*, in which the context is changed to account for the binding of the formals, the execution of the *body*, in which the specification is evaluated, and the *return*, in which the context is changed to account for the binding of the return values.

The effect of the call and return phases depends on the parameter-passing mechanism. In call-by-value, for example, the invocation $r := p(a1, a2, \dots)$ of a procedure p with formals $f1, f2$, etc. is handled by first creating a new context in which the formals are bound to fresh locations in the environment. The values of these locations in the store are copied from the locations of the actuals $a1, a2$, etc. This calling phase is given dependences that map all existing locations to themselves and $f1$ to $a1$, $f2$ to $a2$, etc. The specification is evaluated in the new context. In the return phase, the bindings of the formals are removed from the context, and the location for r acquires the value and the dependences given in the specification for the special variable *result*.

Should the names of the formals coincide with variable names in the calling context, they must be renamed, unless of course they denote global variables. Local static variables of a procedure are handled by treating them as globals that are not accessible to other procedures, with renaming to prevent name clashes.

In value/result parameter passing, the values of the formals' locations are copied back to the actuals in the return phase. For call-by-reference, instead of making a fresh location for each formal, the formal is bound to the location of the actual.

A complication we have ignored arises in the context calculation. Since the model allows multilevel aliasing, the choice of initial context can determine which reconfigurations occur. It is thus, in general, not sufficient either to model a procedure in terms of reconfigurations alone or to check the aliasing effects of a procedure by evaluating it from a single initial context. Elaborating the specification language with context preconditions (declaring expected aliasing patterns at invocation), and adapting the checking mechanism accordingly, would bring little benefit in practice however, since procedures are rarely designed with the assumption of aliasing between arguments.

4. DATA ABSTRACTION: INTRODUCING ASPECTS

With the scheme described so far, the shape of a procedure's specification reflects the type structure of its arguments. Each expression in a dependence or reconfiguration assertion is drawn from the finite set of nodes in the abstract syntax trees of the argument types. In the worst case, therefore, the size of the specification varies with the square of the size of the argument type declarations. But in practice, things are rarely this bad. Few procedures modify more than a handful of components, so most of the dependences are absorbed by the implicit frame condition. Also, components tend to undergo similar treatment, so the shorthand of listing the common dependences of two components together helps significantly. Nevertheless, as types grow, specifications do become more unwieldy.

More seriously, specifications are extremely vulnerable to changes in representation. In a system constructed with abstract types this creates a serious dilemma. Any change in the structure of the representation type invalidates the specifications of the abstract type's procedures, even if the observable behavior is unaltered. Worse, it forces a rewriting of the specifications of external procedures that use the type but have no access to its representation.

The solution adopted in type systems is to hide the structure of an abstract type completely and treat it as a primitive type with no structure. This is no good for Aspect—it would result in specifications so trivial that no bugs would be caught.

4.1 Aspects: Abstract Components

This dilemma is solved by creating a structure for the abstract object that is independent of the structure of its representation. The specifier of an abstract type chooses a set of *aspects*, which are simply names for abstract components. The dependence and reconfiguration assertions are written with aspects instead of concrete names, so clients of the abstract type see only the division into aspects, with the actual representation remaining hidden.

Consider, for example, an abstract type for two-dimensional vectors (Figure 9). The two aspects that are declared, X and Y, view the vector in rectangular terms, although it may well be represented in polar terms. The specifica-

```
abstract type Vector;
    %aspects X, Y;
procedure make (x, y: real): Vector
    %result.X ← x
    %result.Y ← y
procedure add(v1, v2: Vector): Vector
    %result.X ← v1.X, v2.X
    %result.Y ← v1.Y, v2.Y
procedure mag (v: Vector): (real)
    %result ← v.X, v.Y
procedure scale (v: Vector, by: real): Vector
    %result.X ← v.X, by
    %result.Y ← v.Y, by
procedure reflect (v: Vector): Vector
    %result.X ← v.Y
    %result.Y ← v.X
procedure rotate (v: Vector, by: real): Vector
    %result.X, result.Y ← v.X, v.Y, by
```

Fig. 9. The specification of an abstract type with aspects.

tions of the procedures are just like the specifications we have already seen. It is as if the *Vector* type were declared as a record with two fields, X and Y.

Suppose we write a procedure to subtract a vector $v2$ from a vector $v1$:

```
procedure subtractFrom (v1, v2: Vector) is
    %v1.X ← v1.X, v2.X
    %v1.Y ← v1.Y, v2.Y
    v1 := add(v1, scale (v2, −1)).
```

If we thought mistakenly that the *reflect* operation reversed the direction of a vector (when it actually swaps the components) and wrote the body as

```
v1 := add (v1, reflect (v2))
```

the checker would display

```
Missing: v1.X on v2.X
Missing: v1.Y on v2.Y.
```

Aspects have virtually no effect on the checking mechanism. The aspect declaration is treated like a type declaration, with the aspects of an abstract object behaving exactly like the fields of a record. Since the aspect declaration hides the representation, types that sit at the top of a chain of implementations (in which each type is implemented in terms of another) have much smaller structures than they would have without aspects, and checking is much faster.

We mentioned earlier in our discussion of specifications (Section 2.4) how an explicit specification of a called procedure may help detect more bugs, by omitting dependences that are apparent in the called procedure's code. This

happens quite often with aspects. Consider, for example, an alternative implementation that reverses the direction of $v2$ by rotating it:

procedure subtractFrom ($v1$, $v2$: *Vector*)
  $v1 := add(v1, rotate(v2, pi/2))$.

This code has a spurious dependence of $v1.X$ on $v2.Y$ that can be omitted from the specification. Here, the dependence approximation is not to blame. One would need a full specification of *rotate* to determine that the rotation angle in this case happens to avoid any projection of the X component of its argument onto the Y component of its result. The Aspect specification of *subtractFrom* thus embodies the specifier's knowledge of the behavior of *rotate*.

Introducing aspects thus allows the specifier to record details of a procedure's behavior in its specification that could not have been inferred at all from its code. Aspects can also be attributed to primitive types. A file type, for example, might have aspects for the data contents of the file and the position of the file pointer; an integer might have aspects for its sign and magnitude.

4.2 Abstraction Functions

The invention of aspects independent of the actual components of the representation raises an obvious difficulty. How are we now to check the implementation of a procedure of the abstract type? Its specification is cast in terms of aspects which have no obvious correspondence to the actual components of the data representation. What is needed is an association between the two namespaces: the aspects and the actual components. This is provided by an *abstraction function*, specified along with the aspects. Suppose the *Vector* type was implemented in rectangular terms:

$rep = record\ [x, y: real]$.

In this case, the abstraction function would say that the X aspect of an abstract object is obtained from the x component of the corresponding representation object, and similarly Y from y:

abstraction
  $X \ll x$
  $Y \ll y$.

The dependence assertions can then be translated into assertions that can be checked directly. If instead the *Vector* type was represented in polar terms

$rep = record[r, theta: real]$

the abstraction would be

$X \ll r, theta$
$Y \ll r, theta$

indicating that each aspect is obtained from a combination of both representation components. When assertions are translated now, the combination

```
abstract type buffer is
    %aspects text, row, col
    %abstraction
    %      text ≪ lines.@el
    %      row ≪ cy |(pos, lines.@el)
    %      col ≪ cx |(pos, lines.@el)
rep = record [cx, cy: int;
              lines: array of string;
              pos: int]
```

Fig. 10. Abstraction function for a representation with redundancy.

becomes a disjunction. To check the assertion *result.X* ← *v.Y* of *reflect*, for instance, the checker looks for representation dependences of *result.r* or *result.theta* on either *v.r* or *v.theta*. If none of these is present, it displays

Missing (*abs*): *result.X on v.Y*
Missing (*rep*): (*result.r or result.theta*) *on* (*result.r or result.theta*)

showing both the abstract assertion that is violated and the set of acceptable representation dependences that would satisfy it.

Both these representations used concrete types. In general though, one abstract type is implemented in terms of another. The names that appear on the right-hand side of the abstraction function are then aspects rather than concrete components. Aspects may also be used to model pointers to external objects; an abstract set, for example, might have an aspect *elt* pointing to contained objects (see Jackson [1992] for details).

4.3 Redundancy in Representations

Sometimes a representation contains some redundancy, in order to speed up certain operations by trading space for time. Consider an editor buffer, for example, with aspects *text* (for the textual contents), *row* (for the vertical position of the cursor in the text), and *col* (for its horizontal position). It might be represented as an array of strings, one per line (Figure 10). The position in the buffer is held as a simple character count *pos*, so that changes in the buffer at some position can be written through to a file by random access. The cursor, which can be thought of as a row/column pair, is expensive to calculate from this position and the array of strings alone, since it involves adding the line lengths. It therefore makes sense to maintain the cursor redundantly as the two fields *cx* and *cy*.

This redundancy is reflected in the abstraction function. The *row* aspect, for instance, is obtained either from *cy* or from a combination of *pos* and *lines.@el*. Consider now a procedure that gets the column number of the cursor:

```
procedure getCol (b: buffer): int
    %result ← b.col
    return (b.cx).
```

This satisfies its specification, since the checker allows a dependence of *result* on either of the disjuncts. The redundancy in this case weakens the concrete dependency requirements. But in a procedure that modifies the cursor, the redundancy strengthens them. To specify a procedure that adds a character to the end of the last line, for example, we would write:

procedure add (*b*: *buffer*, *c*: *char*)
  %*b.text* ← *b.text*, *c*
  %*b.col* ← *b.col*.

The second assertion now requires two dependences in the representation. If the code inserted the character correctly and updated the *pos* field, but failed to update *cx* in concert, the checker would complain:

Missing (*abs*): *b.col on b.col*
Missing (*rep*): *b.cx on* (*b.cx or b.lines or b.pos*).

The second message says more than could be inferred from the first message and the abstraction function. Some of the representation dependences have been found, but the dependence of *b.cx* is missing. The effect of the dependency analysis here is to ensure that the redundancy is maintained and thus, albeit crudely, to check a representation invariant implicit in the disjunction of the abstraction function.

5. EXPERIENCE

An Aspect checker has been implemented for the CLU programming language [Liskov et al. 1981] and is described in detail in Jackson [1992]. It handles a number of programming language features not described here, such as parametric polymorphism, dynamic allocation, sharing of substructure between abstract objects, and richer control mechanisms (iterators, exceptions, structured jumps, multiple return values). The syntax of the specification language, being tailored to CLU, is slightly different from that presented here.

The checker performs all the checking of dependences, modifications, and reconfigurations described in this article. It also does basic consistency checking of specifications—ensuring, for example, that reconfigurations only relate locations that contain pointers and that when a name like *p.r* appears, the type of *p* admits a field or aspect *r*.

The checker includes a facility for saving libraries of specifications in a compressed form, so that when a procedure is being checked, the specifications of procedures it calls can be loaded quickly. The built-in types (and in fact constructors like records too) are not hardwired but specified in a library that is always loaded. The checker is itself implemented in CLU, in about 15,000 noncomment lines of code.

Aspect has been tested in a series of small case studies. These were of four types, each with its own purpose. First, specifications were written for all the

built-in types of CLU to demonstrate that Aspect is at least expressive enough to describe basic polymorphic types and operations; these specifications totaled about 500 lines.

Second, several toy programs (a few thousand lines long) were written from scratch along with Aspect specifications, to see how Aspect compared to type checking as a technique for detecting flaws in incomplete programs long before testing. Of the bugs missed by the type checker, Aspect caught one bug for every three found later in testing. The ratio of specification to code was almost a half: this was disappointing and led to improvements in the Aspect language; in the final version of the language, the specifications would have been only one quarter of the length of the code.

Third, a few modules (totaling about 6500 lines of code) of an existing program were annotated and checked, to demonstrate that writing and checking Aspect specifications of industrial-quality code are feasible. The LP theorem prover [Garland and Guttag 1989] was chosen: it makes ample use of all the features of CLU, is well documented, and has been used and maintained for several years. Surprisingly, Aspect caught an error that did not cause incorrect behavior but was nevertheless a bug: a procedure's correctness depended on an unstated precondition. The ratio of specs to code was again 1:4.

Fourth, I added code to an existing program (a new help feature for LP) using Aspect to check the new code. About 300 lines of code were written, with about 150 lines of specification, of which about 50 annotated a few hundred lines of existing code. Three bugs were found: one by Aspect and two by testing.

The performance of the checker is adequate for small experiments: checking a procedure of 20 lines with about 100 names (program variables and fields) takes a couple of seconds. The cost of the dependence construction appears to increase linearly with the length of the procedure and quadratically with the number of variables. The worst-case complexity is at least cubic (because of the closure calculation for loops), but this has no practical consequence. Since the analysis is fully modular, the cost is determined only by the size of the largest procedure, not the size of the entire program. A technique for calculating Aspect dependences from the program graph (rather than the syntax tree) has recently been developed [Jackson and Rollins 1994] whose performance is comparable to an optimizing compiler's.

In total, about 3000 lines of Aspect have been written, and about 8000 lines of code have been checked. Experience with Aspect indicates that it can indeed detect bugs missed by the conventional static analyses provided by a compiler and that annotating and checking large programs is feasible. In all the case studies, full specifications were written to detect as many errors as possible. Allowing the checker to generate conservative approximations to missing specifications, and to complete specifications that cover only some arguments of a procedure, would reduce the relative size of the specifications and thus the primary cost of using Aspect.

The kinds of gross error Aspect detects are rare in small programs but a serious problem in very large programs. The most-promising application of

Aspect is likely to be in system integration, where misunderstandings about interfaces are common, and there is a great advantage to any analysis that can precede testing, especially one like Aspect that can incorporate procedures whose code is either inaccessible or unwritten.

6. DISCUSSION AND RELATED WORK

6.1 Anomaly Analysis

Aspect appears to be the first attempt to detect bugs in code by using specifications of dependencies. A similar dependence construction is used in the Spade tool [Bergeretti and Carre 1985] to detect a variety of anomalies, such as the use of a variable before its definition, dead code, or "stable loops" that, from their dependency structure, can be shown to be equivalent to a fixed, finite unfolding. Spade actually uses three dependence relations, of which one (ρ) is the same as the Aspect dependence relation. Other anomaly detectors have used conventional dataflow lattices to determine the possible orders of elementary operations (such as use and definition). A variety of anomalies can be expressed as regular expressions on the language of operations [Fosdick and Osterweil 1976]; the DAVE tool [Osterweil and Fosdick 1976] implemented some important cases.

Some of the assumptions underlying static analysis techniques based on dependences are exposed in Podgurski and Clarke [1990]; they discuss, for example, whether the statement following a loop should have a dependence on the loop test on account of possible nontermination and the resulting consequences on the soundness of various static analyses. Program dependences are investigated semantically by Cartwright and Felleisen [1989], who show that the choice between various dependence notions can be related to the degree of laziness in the store-update operation. A semantic rationale for the kind of dependence used in Aspect has yet to be established, although proving the soundness of the dependence rules (Figure 2) using the definition of dependence given in Section 2.2 is straightforward.

6.2 Dataflow Testing

Dependences have also been used in testing, but for a rather different purpose. Inadequacies in the coverage of a test suite can be exposed, for example, by reporting branches of an if-statement that are never executed—this is branch testing. A more-elaborate coverage criterion can be based on the dependences of the program. One can require, for example, that every def-use association is executed [Rapps and Weyuker 1985]. The costs of dataflow testing are not exhorbitant in practice [Weyuker 1990], nor is it difficult to instrument the program. The difficulty, in common with most test coverage techniques and which rises as coverage criteria become more sophisticated, is eliminating spurious demands (in this case for executing def-use associations that arise in paths that do not correspond to actual program executions). Aspect specifications might have a role here: a dataflow test criterion requiring all specified def-use associations to be covered would not

suffer from the infeasible-paths problem, since, by the specifier's claim, at least one execution satisfying each def-use pair must exist.

6.3 Program Slicing

Program slicing is a transformation based on program dependences. A variable and a program point are selected; a slice of the program is then a new program obtained by deleting statements that preserve the behavior of the original program, for observations of that variable at that point [Weiser 1984]. This notion has found many applications in maintenance, testing, and debugging, and many variants have been proposed [Tip 1994].

The most-popular form of slicing restricts the slice criterion to the selection of a variable that is defined or used at some statement and defines a slice to be a subprogram that includes at least the statements that might affect the value of any of the variables used or defined at that statement [Reps and Yang 1989]. Such a slice is easily computed in the program dependence graph (PDG): it consists of the statements that reach the selected statement via dataflow or control dependence edges [Ottenstein and Ottenstein 1984].

A dependence model that associates Aspect-like dependences with procedures [Jackson and Rollins 1994] can retain the advantages of the program dependence graph (easy construction and slicing as graph traversal) but also allow a simpler treatment of interprocedural slicing than the conventional extension to the PDG [Horwitz et al. 1990] and incorporate external procedures more smoothly.

Moriconi's [1990] analysis for determining the impact of modifications is related to slicing, in that it calculates transitive dependence effects. Its underlying model of dependences is a logic, however; the effects are determined compositionally with an inference rule for each syntactic construct. This analysis like Aspect is modular, and the inferred dependences of a procedure are analogous to an Aspect specification.

6.4 Formal Specification Languages

Like the formal specifications of Larch [Guttag and Horning 1993], VDM [Jones 1990], and Z [Spivey 1992], Aspect specifications are declarative. The order of assertions makes no difference, and they relate only the pre- and poststates of a procedure without prejudicing the internal state transitions. They are "executable" though and, unlike the specifications of these languages, can be checked efficiently. Aspect specifications are, of course, woefully inexpressive in comparison and would certainly not do as contracts between users and implementors.

Whether Aspect specifications are easy enough to write to be cost-effective remains to be seen. What is clear is that conventional specifications are prohibitively expensive and should be reserved primarily for the early stages of the lifecycle and used at the code level only when the correctness of a procedure is so important that the expense is justified (e.g., for component libraries, where the cost is amortized, and for critical components whose failure would be catastrophic).

6.5 Partial Specification Checkers

Developers of regular software need checking methods that lie in cost between type checking and verification. Aspect is one of a number of approaches aimed to fill this gap.

A subset of Larch/C that can be efficiently checked has been implemented in the LCLint tool [Evans et al. 1994]. The modifies clause of a Larch/C specification requires that at most some set of variables be modified. LCLint, in contrast to Aspect (which would check for plausible modifications of those variable) checks that no other variables are modified. It also detects violations of declared abstraction barriers, which are not necessarily bugs but are often symptoms of (sometimes quite subtle) errors.

Henderson's [1975] work on finite-state modeling of programs would now be classified as abstract interpretation, even though it predates the work of Cousot and Cousot [1977]. A program is constructed in layers, with each implementing an abstract machine. The procedures of each layer are specified by giving transitions in an abstract state space invented by the specifier; from these specifications, the transition functions of the procedures of upper layers are constructed and compared to their specifications. More recently, the Syntox tool checks assertions in Pascal programs by approximating the reachable states at each program point with, roughly speaking, an abstract calculation of the weakest precondition [Bourdoncle 1993]. By exploiting more-elaborate lattices of abstract values, along with widening and narrowing—ad hoc means for obtaining convergence along infinite chains—Syntox is able to detect some very subtle bugs, albeit in small numerical programs.

A number of techniques have been developed to detect cases in which operations are applied to an object in an illegal order. The typestates of the Nil programming language [Strom 1983] distinguish, in the type system, the states of a pointer before and after initialization, and by enforcing a requirement that all apparent paths to a program point give the same approximated typestates, they can prevent dereferencing of uninitialized pointers. Refinement types [Freeman and Pfenning 1991] extend the idea to higher-order functions in ML: a recursive type definition may be written that allows richer types to be inferred for the constructors, so that subsequent type inference can catch more errors. The list type, for instance, can be enriched to distinguish the empty, singleton, and longer lists.

The event sequence analysis of Cesar/Cecil [Olender and Osterweil 1992] lets the user specify explicitly for each datatype the sequences of operations that are acceptable. A regular expression can express, for example, the constraint that a read-only file must be opened, read zero or more times, and then closed.

Perry's Inscape environment [Perry 1989a] centers around "constructive use" of specifications, which includes the detection of bugs. The transitions of a procedure are specified by propositional assertions: for example, one might assert that the *close* procedure on the file f can only be executed when *Open*(f) is true, and subsequently *Closed*(f) is true. In addition to pre- and postconditions, a specification may include "postobligations"; for the *open*

procedure, we would assert that it must eventually be closed. A novel propositional logic is used to propagate assertions around the code of a procedure [Perry 1989b]. Errors are reported when it can be shown that an assertion is violated. If a *read* of a file immediately follows a close, for example, its precondition—that the file be open—cannot be satisfied, and there must be a bug.

All these state-based schemes identify errors with execution paths that can produce bad values, so they tend to be good at detecting errors of commission. Aspect, on the other hand, looks for missing paths and is good at errors of omission. It is encouraging that the techniques seem to be complementary.

Howden's [1990] "comments analysis" is similar to these techniques, in that it associates assertions, called *comments*, with program points. Roughly, a comment is a propositional statement about the program variables. The proposition names are invented by the specifier and thus might be similar to the assertions of Inscape or the typestates of Nil. But a proposition can express more than the program state. A *flavor assertion* can make distinctions like an abstract type system with name equality: for an integer i, for instance, we might have *isTemperature*(i) and *isLength*(i). Alternatively, a flavor assertion may indicate the computation to be expected—*isTotalSum*(i) vs. *isPartialSum*(i)—and might even incorporate elements of program dependence—*dependsOnX*(y). The Quick Defect Analyzer (QDA) [Howden and Wieand 1994] was designed to perform a comments analysis on an avionics program written in assembler and was extremely successful in detecting bugs, even though the program had been running for some time. QDA's basic function is to propagate flavor assertions along edges of the control-flow graph, checking the flavors hypothesized at one program point against flavors asserted at another. In addition, "rules" allow flavors of different variables to be related across the entire program. One can assert, for example, that some register holds the value 5 when the nose of the airplane is down. Many errors were found because different procedures declared inconsistent rules (and thus interpreted data incompatibly). Rules can also play a role similar to Aspect's abstraction function, translating assertions about the detailed representation of data into assertions about its abstract properties.

6.6 Alias Analysis

There is a long history of work in alias analysis; for the most-recent contributions see Choi et al. [1993], Deutsch [1994], and Pande et al. [1994]. Aspect's context calculation is not intended to make novel contributions in this area—indeed it is similar to schemes based on abstract interpretation, such as Larus and Hilfinger [1988] and Horwitz et al. [1989]—but rather to provide a minimal treatment of aliasing that is sufficient, in practice, to allow accurate construction of dependences. Aspect, moreover, is constrained in a way conventional alias analyses are not. Its representation of aliasing must not only be computable and sound but also specifiable. It is for this reason that Aspect does not handle recursive structures such as linked lists and trees. So long as they are encapsulated within an abstract type that can be

viewed as a collection, they can be specified and their clients analyzed, but the code that manipulates the actual linked structures is not amenable to Aspect checking. These structures are, incidentally, rare in CLU programs, since the built-in dynamic array type provides better performance than linked lists and gives much cleaner code. Perhaps Aspect is worth extending, but I suspect that the burden of specification may be excessive.

The alias analysis scheme, even within the model presented here, could be improved. There are circumstances in which patently unnecessary dependences are inserted. The checker actually performs a better analysis than the analysis described here, but it is unclear whether the complication pays off in practice. Indeed, reconfigurations are usually far harder to specify than dependences, and it may be worth weakening the alias analysis to reduce the burden of specification. Schemes that divide the store into regions of potentially aliased locations [Gifford and Lucassen 1988] are much simpler and may be adequate in some applications. They are also more easily extended to higher-order functions.

APPENDIX

A. THE ASPECT LANGUAGE

This grammar gives the syntax of the subset of the language explained in this article. The expression $S,^*$ ($S,^+$) denotes zero (one) or more iterations of S, separated by commas.

procedure-annotation: := (dependency | reconfiguration)*
dependency: := expr "←" expr,*
reconfiguration: := ptr ":-" ptr,*
expr: := obj-name[.aspect]
obj-name: := variable | variable[.fields]
fields: := (field-name | field-name "↑" | "@el").*
type-annotation: := "aspects" aspect,* "abstraction" abst-decl*
abst-decl: := aspect ≪ conc-expr,+
conc-expr: := fields[.aspect]("|" fields[.aspect])*

B. MATHEMATICAL SYMBOLS

$\mathsf{P}\,S$ powerset of S
$S \rightarrow T$ functions from S to T
$S \leftrightarrow T$ binary relations from S to T
$\#S$ size of set S
$x \mapsto y$ maplet, equivalent to pair (x, y)
$I_S = \{(x, x) \mid x \in S\}$ identity relation over set S
$R[S] = \{y \mid \exists x \in S.\,(x, y) \in R\}$ image of set S under relation R

$Q \circ R = \{(x, z) \mid \exists y.\ (x, y) \in Q \wedge (y, z) \in R\}$ relational product
$R^{+} = R \cup (R \circ R) \cup \ldots$ transitive closure of R
$R^{*} = I \cup R \cup (R \circ R) \cup \ldots$ reflexive and transitive closure of R
$Q \oplus R = \{(x, y) \mid (x, y) \in R \vee ((x, y) \in Q \wedge \nexists z.\ (x, z) \in R)\}$ relational override
$\langle\rangle$ empty sequence
$\langle e \rangle$ sequence containing single element e
$s \frown t$ concatenation of sequences s and t

ACKNOWLEDGMENTS

John Guttag supervised the thesis on which this article is based. He contributed greatly to the emphasis of the work, notably in the role of abstraction and the declarative nature of the specifications. Butler Lampson complained incessantly about Aspect's complexity (with some effect, I hope) and persuaded me to recast Aspect in a more-traditional Algol-like model. David Gifford gave me helpful feedback, especially early on, and Pamela Zave gave me the benefits of her careful reading and sound advice on presentation. Keith Randall was, after me, Aspect's first user and the implementor of the second version of the Aspect Checker, with help from Dorothy Curtis. I also received valuable comments from John Gannon, Michael Jackson, Rustan Leino, Gail Murphy, Robert O'Callahan, Mark Reinhold, Mark Vandevoorde, and the anonymous referees. Many thanks to them all.

REFERENCES

BERGERETTI, J. F. AND CARRE, B. A. 1985. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan.).

BOURDONCLE, F. 1993. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM Symposium of Programming Language Design and Implementation*. ACM, New York.

CHOI, J., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*. ACM, New York, 232–245.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*. ACM, New York, 238–252.

CARTWRIGHT, R. AND FELLEISEN, M. 1989. The semantics of program dependence. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*. ACM, New York.

DEUTSCH, A. 1994. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN T94 Conference on Programming Language Design and Implementation*. ACM, New York, 230–241.

EVANS, D., GUTTAG, J., HORNING, J., AND TAN, Y. M. 1994. LCLint—a tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Conference on Foundations of Software Engineering* (New Orleans, La., Dec.). ACM, New York.

FOSDICK, L. D. AND OSTERWEIL, L. J. 1976. Dataflow analysis in software reliability. *ACM Comput. Surv. 8*, 3 (Sept.).

FREEMAN, T. AND PFENNING, F. 1991. Refinement types for ML. In *Proceedings of the ACM Conference on Principles of Programming Languages*. ACM, New York.

GARLAND, S. J AND GUTTAG, J V. 1989. An overview of LP, the Larch Prover. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*. Lecture Notes in Computer Science, vol. 355. Springer-Verlag, New York

GIFFORD, D. K. AND LUCASSEN, J. M. 1988. Polymorphic effect systems. In the *ACM Symposium on Principles of Programming Languages* ACM, New York.

GUTTAG, J. V. AND HORNING, J. J. 1993. *Larch: Languages and Tools for Formal Specifications*. Springer-Verlag, New York.

HENDERSON, P. 1975. Finite state modelling in program development. In *Proceedings of the International Conference on Reliable Software*. ACM, New York

HOWDEN, W. E. 1990. Comments analysis and programming errors. *IEEE Trans. Softw. Eng. 16*, 1 (Jan.).

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12*, 1 (Jan.), 26–60.

HORWITZ, S., PFEIFFER, P., AND REPS, T. 1989. Dependency analysis for pointer variables. In the *ACM Symposium on Principles of Programming Language Design and Implementation*. ACM, New York.

HOWDEN, W. E. AND WIEAND, B. 1994. QDA—a method for systematic informal program analysis. *IEEE Trans. Softw. Eng. 20*, 6 (June).

JACKSON, D. 1992. Aspects. A formal specification language for detecting bugs. Tech. Rep. MIT/LCS/TR-543, MIT Laboratory for Computer Science, Cambridge, Mass. June.

JACKSON, D. AND ROLLINS, E. J. 1994. A new model of program dependences for reverse engineering. In *Proceedings of the 2nd ACM SIGSOFT Conference on Foundations of Software Engineering* (New Orleans, La., Dec.). ACM, New York.

JONES, C. 1990. *Systematic Software Development Using VDM*, 2nd ed. Prentice-Hall, Englewood Cliffs, N.J.

LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflicts between structure accesses. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, New York.

LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SCHAFFERT, C., SCHEIFLER, B., AND SNYDER, A. 1981. *CLU Reference Manual*. Springer-Verlag, Berlin.

MORICONI, M. 1990. Approximate reasoning about the semantics effects of program changes. *IEEE Trans. Softw. Eng. 16*, 9 (Sept.).

OSTERWEIL, L. J. AND FOSDICK, L. D. 1976. DAVE—a validation, error detection and documentation system for Fortran programs. *Softw. Pract. Exper. 6*, 4, 473–486.

OTTENSTEIN, K. J. AND OTTENSTEIN, L. M. 1984. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments* *ACM SIGPLAN Not. 19*, 5 (May), 177–184.

OLENDER, K. M. AND OSTERWEIL, L. J. 1992. Interprocedural static analysis of sequencing constraints. *ACM Trans. Softw. Eng. Method. 1*, 1 (Jan.).

PODGURSKI, A. AND CLARKE, L. 1990. A formal model of program dependencies and its implications for software testing, debugging and maintenance. *IEEE Trans. Softw. Eng. 16*, 9 (Sept.).

PERRY, D. E. 1989a. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering* (Pittsburgh, Pa., May). ACM, New York, 2–12.

PERRY, D. E. 1989b. The logic of propagation in the Inscape environment. In *Proceedings of the 3rd ACM Symposium on Software Testing, Analysis and Verification (TAV3)* (Key West, Fla., Dec.). *ACM Softw. Eng. Not. 14*, 8 (Dec.), 114–121.

PANDE, H., LANDI, W., AND RYDER, B. 1994. Interprocedural def-use associations for C systems. *IEEE Trans. Softw. Eng. 20*, 5 (May), 385–403.

RAPPS, S. AND WEYUKER, E. J. 1985. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr.), 367–375.

RUSSELL, J. R., STROM, R. E, AND YELLIN, D. M. 1994 A checkable interface language for pointer-based structures. In *Proceedings of the Workshop on Interface Definition Languages*. *ACM SIGPLAN Not. 29*, 8 (Aug.).

REPS, T. AND YANG, W. 1989. The semantics of program slicing and program integration. In *Proceedings of the Colloquium on Current Issues in Programming Languages* (Barcelona, Spain, Mar.). Lecture Notes in Computer Science, vol. 352. Springer-Verlag, New York, 360–374.

SPIVEY, J. M. 1992. *The Z Notation: A Reference Manual.* 2nd ed. Prentice-Hall, Englewood Cliffs, N.J.

STROM, R. E. 1983. Mechanisms for compile-time enforcement of security. In *Proceedings of the ACM Conference on Principles of Programming Languages.* ACM, New York.

TIP, F 1994. A survey of program slicing techniques. Tech. Rep. CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10*, 4 (July).

WEYUKER, E. J. 1990. The cost of data flow testing: An empirical study. *IEEE Trans. Softw. Eng. 16*, 2 (Feb.).

Received November 1993; revised March 1995; accepted June 1995